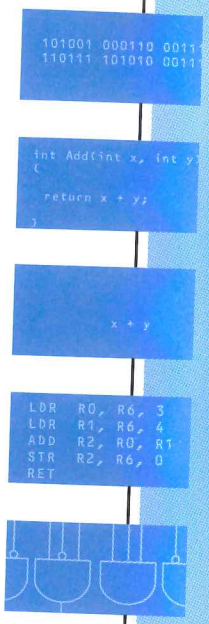
chapter

19



Data Structures

19.1 Introduction

C, at its core, provides support for three fundamental types of data: integers, characters, and floating point values.[1] That is, C natively supports the allocation of variables of these types and natively supports operators that manipulate these types, such as + for addition and * for multiplication. As we traversed the topics in the second half of this textbook, we saw the need for extending these basic types to include pointers and arrays. Both pointers and arrays are derived from the three fundamental types. Pointers point to one of the three types; we can declare arrays of `int`, `char`, or `double`.

Ultimately, though, the job of the programmer is to write programs that deal with real-world objects, such as an aircraft wing or a group of people or a pod of migrating whales. The problem lies in the reality that integers, characters, and floating point values are the only things that the underlying computing system can deal with. The programmer must map these real-world objects onto these primitive types, which can be burdensome. But the programming language can assist in making that bridge. Providing support for describing real-world objects and specifying operations upon them is the basis for *object orientation*.

Orienting a program around the objects that it manipulates rather than the primitive types that the hardware supports is the basic precept of object-oriented programming. We take a small step toward object orientation in this chapter by examining how a C programmer can build a type that is a combination of the

[1] Enumerations are another fundamental type that are closely tied to integer types.

more basic types. This aggregation is called a *structure* in C. Structures provide the programmer with a convenient way of representing objects that are best represented by multiple values. For example, an employee might be represented as a structure containing a name (character string), job title (character string), department (perhaps integer), and employee ID (integer) within a corporate database program. In devising such a database program we might use a C structure.

The main theme of this chapter is C's support for advanced data structures. First, we examine how to create structures in C and examine a simple program that manipulates an array of structures. Second, we examine dynamic memory allocation in C. Dynamic allocation is not directly related to the concept of structures, but it is a component we use for the third item of this chapter, linked lists. A linked list is a fundamental (and common) data organization that is similar to an array—both store collections of data items—but has a different organization for its data items. We will look at functions for adding, deleting, and searching for data items within linked lists.

19.2 Structures

Some things are best described by an aggregation of fundamental types. For such objects, C provides the concept of structures. Structures allow the programmer to define a new type that consists of a combination of fundamental data items such as `int`, `char`, and `double`, as well as pointers to them and arrays of them. Structure variables are declared in the same way variables of fundamental data types are declared. Before any structure variables are declared, however, the organization and naming of the data items within the structure must be defined.

For example, in representing an airborne aircraft, say for a flight simulator or for a program that manages air traffic over Chicago, we would want to describe several flight characteristics that are relevant for the application at hand. The aircraft's flight number is useful for identification, and since this would typically be a sequence of digits and characters, we could use a character string for representing it. The altitude, longitude, latitude, and heading of the flight are also useful, all of which we might store as integers. Airspeed is another characteristic that would be important, and it is best represented as a double-precision floating point number. Following are the variable declarations for describing a single aircraft in flight.

```
char flightNum[7];   /* Max 6 characters          */
int altitude;        /* in meters                 */
int longitude;       /* in tenths of degrees      */
int latitude;        /* in tenths of degrees      */
int heading;         /* in tenths of degrees      */
double airSpeed;     /* in kilometers/hour        */
```

If the program modeled multiple flights, we would need to declare a copy of these variables for each one, which is tedious and could result in excessively long code. C provides a convenient way to aggregate these characteristics into a single type via the `struct` construct, as follows:

```
struct flightType {
  char flightNum[7];  /* Max 6 characters          */
  int altitude;       /* in meters                 */
  int longitude;      /* in tenths of degrees      */
  int latitude;       /* in tenths of degrees      */
  int heading;        /* in tenths of degrees      */
  double airSpeed;    /* in kilometers/hour        */
};
```

In the preceding declaration, we have created a new type containing six *member* elements. We have not yet declared any storage; rather we have indicated to the compiler the composition of this new type. We have given the structure the *tag* `flightType`, which is necessary for referring to the structure in other parts of the code.

To declare a variable of this new type, we do the following:

```
struct flightType plane;
```

This declares a variable called `plane` that consists of the six fields defined in the structure declaration but otherwise gets treated like any other variable.

We can access the individual members of this structure variable using the following syntax:

```
struct flightType plane;

plane.airSpeed = 800.00;
plane.altitude = 10000;
```

Each member can be accessed using the variable's name as the base name followed by a dot `.` followed by the member name.

The variable declaration `plane` gets allocated onto the stack if it is a local variable and occupies a contiguous region of memory large enough to hold all member elements. In this case, if each of the fundamental types occupies one LC-3 memory location, the variable `plane` would occupy 12 locations.

The allocation of the structure is straightforward. A structure is allocated the same way a variable of a basic data type is allocated: locals (by default) are allocated on the run-time stack, and globals are allocated in the global data section. Figure 19.1 shows a portion of the run-time stack when a function that contains the following declarations is invoked.

```
int x;
struct airplaneType plane;
int y;
```

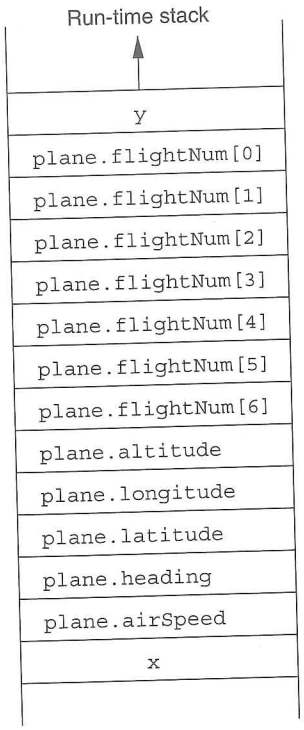


Figure 19.1 The run-time stack showing an allocation of a variable of structure type

Generically, the syntax for a structure declaration is as follows:

```
struct tag {
  type1 member1;
  type2 member2;
  ...
  typeN memberN
} identifiers;
```

The `tag` provides a handle for referring to the structure later in the code, as in the case of later declaring variables of the structure's format. The list of members defines the organization of a structure and is syntactically a list of declarations. A member can be of any type, including another structure type. Finally, we can optionally include identifiers in a structure's declaration to actually declare variables of that structure's type. These appear after the closing brace of the structure declaration, prior to the semicolon.

19.2.1 `typedef`

C structures enable programmers to define their own types. C `typedef` allows programmers to name their own types. It has the general form

```
typedef type name;
```

This statement causes the identifier `name` to be synonymous with the type `type`, which can be any basic type or aggregate type (e.g., a structure). So for instance,

```
typedef int Color;
```

allows us to define variables of type `Color`, which will now be synonymous with integer. Using this definition, we can declare (for a bitmapped image, for example):

```
Color pixels[500];
```

The `typedef` declaration is particularly useful when dealing with structures. For example, we can create a name for the structure we defined earlier:

```
struct flightType {
  char flightNum[7];   /* Max 6 characters          */
  int altitude;        /* in meters                 */
  int longitude;       /* in tenths of degrees      */
  int latitude;        /* in tenths of degrees      */
  int heading;         /* in tenths of degrees      */
  double airSpeed;     /* in kilometers/hour        */
};

typedef struct flightType Flight;
```

Now we can declare variables of this type by using the type name `Flight`. For example,

```
Flight plane;
```

is now equivalent to the declaration `struct flightType plane;` that we used previously.

The `typedef` declaration provides no additional functionality. However, it gives clarity to code, particularly code heavy with programmer-defined types. Well-chosen type names connote properties of the variables they declare even beyond what can be expressed by the names of the variables themselves.

19.2.2 Implementing Structures in C

Now that we have seen the technique for declaring and allocating variables of structure type (and have given them new type names), we focus on accessing the member fields and performing operations on them. For example, in the following code, the member `altitude` of the structure variable of type `Flight` is accessed.

```
int x;
Flight plane;
int y;

plane.altitude = 0;
```

Here, the variable `plane` is of type `Flight`, meaning it contains the six member fields we defined previously. The member field labeled `altitude` is accessed using the variable's name followed by a period, followed by the member field label. The compiler, knowing the layout of the structure, generates code that accesses the structure's member field using the appropriate offset. Figure 19.1 shows the layout of the portion of the activation record for this function. The compiler keeps track, in its symbol table, of the position of each variable in relation to the base pointer R5, and if the variable is an aggregate data type, it also tracks the position of each field within the variable. Notice that for the particular reference `plane.altitude = 0;`, the compiler must generate code to access the second variable on the stack and the second member element of that variable.

Following is the code generated by the LC-3 C compiler for the assignment statement `plane.altitude = 0;`.

```
AND   R1, R1, #0    ; zero out R1

ADD   R0, R5, #-12  ; R0 contains base address of plane
STR   R1, R0, #7    ; plane.altitude = 0;
```

19.3 Arrays of Structures

Let's say we are writing a piece of software to determine if any flights over the skies of Chicago are in danger of colliding. For this program, we will use the `Flight` type that we previously defined. If the maximum number of flights that will ever simultaneously exist in this airspace is 100 planes, then the following declaration is appropriate:

```
Flight planes[100];
```

This declaration is similar to the simple declaration `int d[100]`, except instead of declaring 100 integer values, we have declared a contiguous region of memory containing 100 structures, each of which is composed of the six members indicated in the declaration `struct flightType`. The reference `planes[12]`, for example, would refer to the thirteenth object in the region of 100 such objects in memory. Each object contains enough storage for its six constituent member elements.

Each element of this array is of type `Flight` and can be accessed using standard array notation. For example, accessing the flight characteristics of the first flight can be done using the identifier `plane[0]`. Accessing a member field is done by accessing an element of the array and then specifying a field: `plane[0].heading`. The following code segment provides an example. It finds the average airspeed of all flights in the airspace monitored by the program.

```
int i;
double sum = 0;
double averageAirSpeed;

for (i = 0; i < 100; i++)
   sum = sum + plane[i].airSpeed;

averageAirSpeed = sum / 100;
```

We can also create pointers to structures. The following declaration creates a pointer variable that contains the address of a variable of type `Flight`.

```
Flight *planePtr;
```

We can assign this variable as we would any pointer variable.

```
planePtr = &plane[34];
```

If we want to access any of the member fields pointed to by this pointer variable, we could use an expression such as the following:

```
(*planePtr).longitude
```

With this cumbersome expression, we are dereferencing the variable `planePtr`. It points to something of type `Flight`. Therefore when `planePtr` is dereferenced, we are accessing an object of type `Flight`. We can access one of its member fields by using the dot operator (`.`). As we shall see, refering to a structure with a pointer is a common operation, and since this expression is not very straightforward to grasp, a special operator has been defined for it. The previous expression is equivalent to

```
planePtr->longitude
```

That is, the expression `->` is like the deference operator `*`, except it is used for deferencing member elements of a structure type.

Now we are ready to put our discussion of structures to use by presenting an example of a function that manipulates an array of structures. This example examines the 100 flights that are airborne to determine if any pair of them are potentially in danger of colliding. To do this, we need to examine the position, altitude, and heading of each flight to determine if there exists the potential of collision. In Figure 19.2, the function `PotentialCollisions` calls the function `Collide` on each pair of flights to determine if their flight paths dangerously intersect. (This function is only partially complete; it is left as an exercise for you to write the code to more precisely determine if two flight paths intersect.)

Notice that `PotentialCollisions` passes `Collide` two pointers rather than the structures themselves. While it is possible to pass structures, passing pointers is likely to be more efficient because it involves less pushing of data onto the run-time stack; that is, in this case two pointers are pushed rather than 24 locations' worth of data for two objects of type `Flight`.

```
#include <stdio.h>
#define TOTAL_FLIGHTS 100

/* Structure definition */
struct flightType {
  char flightNum[7];   /* Max 6 characters          */
  int altitude;        /* in meters                 */
  int longitude;       /* in tenths of degrees      */
  int latitude;        /* in tenths of degrees      */
  int heading;         /* in tenths of degrees      */
  double airSpeed;     /* in kilometers/hour        */
};

typedef struct flightType Flight;

int Collide(Flight *planeA, Flight *planeB); void
PotentialCollisions(Flight planes[]);

int Collide(Flight *planeA, Flight *planeB)
{
  if (planeA->altitude == planeB->altitude) {

    /** More logic to detect collision goes here **/
  }
  else
    return 0;
}

void PotentialCollisions(Flight planes[])
{
  int i;
  int j;

  for (i = 0; i < TOTAL_FLIGHTS; i++) {
    for (j = 0; j < TOTAL_FLIGHTS; j++) {
      if (Collide(&planes[i], &planes[j]))
        printf("Flights %s and %s are on collision course!\n",
               planes[i].flightNum, planes[j].flightNum);
    }
  }
}
```

Figure 19.2 An example function based on the structure `Flight`

19.4 Dynamic Memory Allocation

Memory objects (e.g., variables) in C programs are allocated to one of three spots in memory: the run-time stack, the global data section, or the *heap*. Variables declared local to functions are allocated during execution onto the run-time stack by default. Global variables are allocated to the global data section and are

accessible by all parts of a program. Dynamically allocated data objects—objects that are created during run-time—are allocated onto the heap.

In the previous example, we declared an array that contained 100 objects, where each object was an aircraft in flight. But what if we wanted to create a flexible program that could handle as many flights as were airborne at any given moment, whether it be 2 or 20,000? One possible solution would be to declare the array assuming a large upper limit to the number of flights the program might encounter. This could result in a lot of potentially wasted memory space, or worse, we might underestimate the number of flights, which could have potentially devastating repercussions. A better solution is to dynamically adapt the size of the array based on the number of planes in the air. To accomplish this, we rely on the concept of dynamic memory allocation.

In a nutshell, dynamic memory allocation works as follows: A piece of code called the memory allocator manages an area of memory called the heap. Figure 19.3 is a copy of Figure 12.7; it shows the relationship of the various regions of memory, including the heap. During execution, a program can make requests to the memory allocator for contiguous pieces of memory of a particular

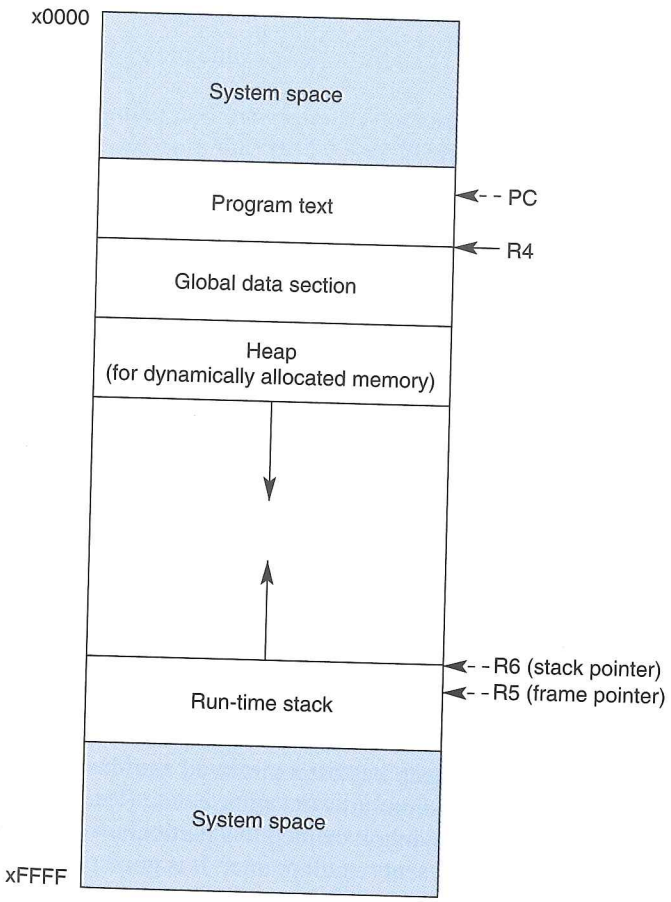


Figure 19.3 The LC-3 memory map showing the heap region of memory

size. The memory allocator then reserves this memory and returns a pointer to the newly reserved memory to the program. For example, if we wanted to store 1,000 flights' worth of data in our air traffic control program, we could request the allocator for this space. If enough space exists in the heap, the allocator will return a pointer to it. Notice that the heap and the stack both grow toward each other. The size of the stack is based on the depth of the current function call, whereas the size of the heap is based on how much memory the memory allocator has reserved for the requests it has received.

A block of memory that is allocated onto the heap stays allocated until the programmer explicitly deallocates it by calling the memory deallocator. The deallocator adds the block back onto the heap for subsequent reallocation.

19.4.1 Dynamically Sized Arrays

Dynamic allocation in C is handled by the C standard library functions. In particular, the memory allocator is invoked by the function `malloc`. Let's take a look at an example that uses the function `malloc`:

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes = malloc(24 * airbornePlanes);
```

The function `malloc` allocates a contiguous region of memory on the heap of the size in bytes indicated by the single parameter. If the heap has enough unclaimed memory and the call is successful, `malloc` returns a pointer to the allocated region.

Here we allocate a chunk of memory consisting of `24 * airbornePlane` bytes, where `airbornePlanes` is the number of planes in the air as indicated by the user. What about the 24? Recall that the type `Flight` is composed of six members—an array of 7 characters, 4 integers, and a double, each occupy a single two-byte location on the LC-3. Each structure requires 24 bytes of memory. As a necessary convenience for programmers, the C language supports a compile-time operator called `sizeof`. This operator returns the size, in bytes, of the memory object or type passed to it as an argument. For example, `sizeof(Flight)` will return the number of bytes occupied by a variable of type `Flight`, or 24. The programmer does not need to calculate the sizes of various data objects; the compiler can be instructed to perform the calculation.

If all the memory on the heap has been allocated and the current allocation cannot be accomplished, `malloc` returns the value `NULL`. Recall that the symbol `NULL` is a preprocessor macro symbol, defined to a particular value depending on the computer system, that represents a null pointer. It is good programming practice to check that the return value from `malloc` indicates the memory allocation was successful.

The function `malloc` returns a pointer. But what is the type of the pointer? In the preceding example, we are treating the pointer that is returned by `malloc` as a pointer to some variable of type `Flight`. Later we might use `malloc` to allocate an array of integers, meaning the return value will be treated as an `int *`. To enable this, `malloc` returns a generic data pointer, or `void *`, that needs to be *type cast* to the appropriate form upon return. That is, whenever we call the memory allocator, we need to instruct the compiler to treat the return value as of a *different* type than was declared.

In the preceding example, we need to type cast the pointer returned by `malloc` to the type of the variable to which we are assigning it. Since we assigned the pointer to `planes`, which is of type `Flight *`, we therefore cast the pointer to type `Flight *`. To do otherwise makes the code less portable across different computer systems; most compilers generate a warning message because we are assigning a pointer value of one type to a pointer variable of another. Type casting causes the compiler to treat a value of one type as if it were of another type. To type cast a value from one type to a `newType`, we use the following syntax. The variable `var` should be of `newType`. For more information on type casting, refer to section D.5.11.

```
var = (newType) expression;
```

Given type casting and the `sizeof` operation and the error checking of the return value from `malloc`, the correct way to write the code from the previous example is:

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

/* A more correctly written call malloc */
planes = (Flight *) malloc(sizeof(Flight) * airbornePlanes);
if (planes == NULL) {
   printf("Error in allocating the planes array\n");
   :
   :
}
plane[0].altitude = ...
```

Since the region that is allocated by `malloc` is contiguous in memory, we can switch between pointer notation and array notation. Now we can use the expression `planes[29]` to access the characteristics of the 30th aircraft (provided that `airbornePlanes` was larger than 30, of course). Notice that we smoothly switched from pointer notation to array notation; this flexibility has helped make C a very popular programming language. Other derivative languages, C++ in particular, keep this duality between pointers to contiguous memory and arrays.

The function `malloc` is only one of several memory allocation functions in the standard library. The function `calloc` allocates memory and initializes it to the

value 0. The function `realloc` attempts to grow or shrink previously allocated regions of memory. To use the memory allocation functions of the C standard library, we need to include the `stdlib.h` header file. Can you use `realloc` to create an array that adapts to the size of the data size—for example, write a function `AddPlane()` that adds a plane if the current size of the `planes` is too small? Likewise, write the function `DeletePlane()` when the size of the array is larger than what is required.

A very important counterpart to the memory allocation functions is a function to *deallocate* memory and return it to the heap. This function is called `free`. It takes as its parameter a pointer to a region that was previously allocated by `malloc` (or `calloc` or `realloc`) and deallocates it. After a region has been `free`'d, it is once again eligible for allocation. Why is deallocation necessary? As we shall see, there is a class of data structures that dynamically grow and shrink as the program executes. For the shrinking operation, we put allocated memory back on the heap so that we can use it again in subsequent allocations.

19.5 Linked Lists

Having discussed the notion of structures and the concept of dynamic memory allocation, we are now ready to introduce a fundamental data structure that is pervasive in computing. A *linked list* is similar to an array in that both can be used to store data that is best represented as a list of elements. In an array, each element (except the last) has a next element that follows it sequentially in memory. Likewise in a linked list, each element has a next element, but the next element need not be sequentially adjacent in memory. Rather, each element contains a pointer to the next element.

A linked list is a collection of *nodes*, where each node is one "unit" of data, such as the characteristics of an airborne aircraft from the previous section. In a linked list we connect these nodes together using pointers. Each node contains a pointer element that points to the next node in the list. Given a starting node, we can go from one node to another by following the pointer in each node. To create these nodes, we rely on C structures. A critical element for the structure that defines the nodes of a linked list is that it contains a member element that points to nodes like itself. The following code demonstrates how this is accomplished. We use the `Flight` type we defined in the previous sections. Notice that we have added a new member element to the structure definition. It is a pointer to a node of the same type.

```
typedef struct flightType Flight;
struct flightType {
  char flightNum[7];   /* Max 6 characters          */
  int altitude;        /* in meters                 */
  int longitude;       /* in tenths of degrees      */
  int latitude;        /* in tenths of degrees      */
  int heading;         /* in tenths of degrees      */
  double airSpeed;     /* in kilometers/hour        */
  Flight *next;
};
```

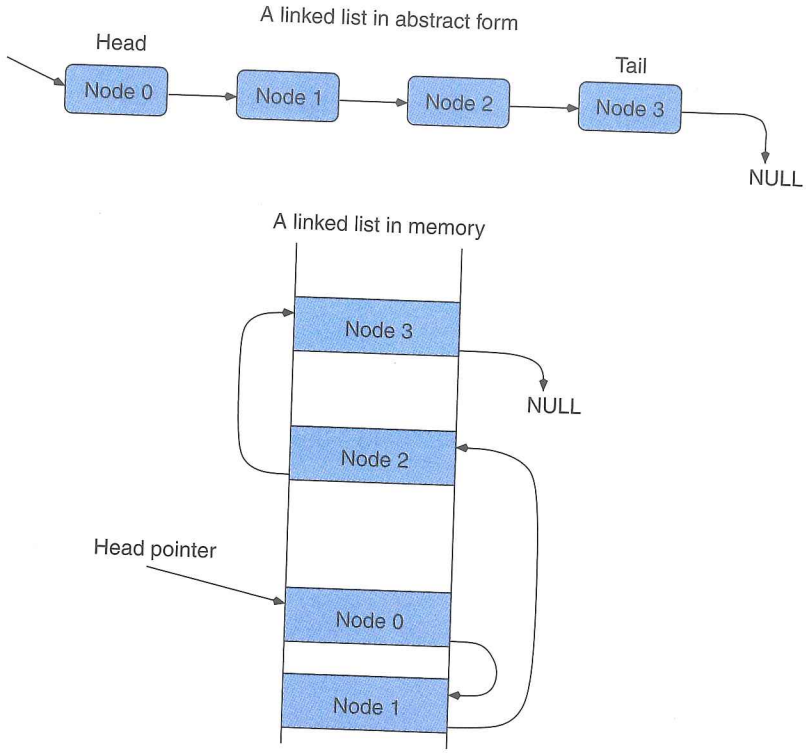


Figure 19.4 Two representations for a linked list

Like an array, a linked list has a beginning and an end. Its beginning, or *head*, is accessed using a pointer called the *head pointer*. The final node in the list, or tail, points to the `NULL` value. Figure 19.4 shows two representations of a linked list data structure: an abstract depiction where nodes are represented as blocks and pointers are represented by arrows, and a more physical representation that shows what the data structure might look like in memory.

Despite their similarities, arrays and linked lists have fundamental differences. An array can be accessed in random order. We can access element number 4, followed by element 911, followed by 45, for example. A simple linked list must be traversed sequentially starting at its head. If we wanted to access node 29, then we must start at node 0 (the head node) and then go to node 1, then to node 2, and so forth. But linked lists are dynamic in nature; additional nodes can be added or deleted without movement of the other nodes. While it is straightforward to dynamically size an array (see Section 19.4.1 on using `malloc`), it is much more costly to remove a single element in an array, particularly if it lies in the middle. Consider, for example, how you would remove the information for a plane that has just landed from the air traffic control program from Section 19.3. With a linked list we can dynamically add nodes to make room for more data, and we can delete nodes that are no longer required.

19.5.1 An Example

Say we want to write a program to manage the inventory at a used car lot. At the lot, cars keep coming and going, and the database needs to be updated continually—a new entry is created whenever a car is added to the lot and an entry deleted whenever a car is sold. Furthermore, the entries are stored in order by vehicle identification number so that queries from the used car salespeople can be handled quickly. The information we need to keep per car is as follows:

```
int  vehicleID;    /* Unique identifier for a car */
char make[20];     /* Manufacturer                */
char model[20];    /* Model name                  */
int  year;         /* Year of manufacture         */
int mileage;       /* in miles                    */
double cost;       /* in dollars                  */

Car *next;         /* Points to a car_node        */
```

In reality, a vehicle ID is a sequence of characters and numbers and cannot be stored as a single `int`, but we store it as an integer to make the example simpler.

The frequent operations we want to perform—adding, deleting, and searching for entries—can be performed simply and quickly using a linked list data structure. Each node in the linked list contains all the information associated with a car in the lot, as shown. We can now define the node structure, which is then given the name `CarNode` using `typedef`:

```
typedef struct carType Car;

struct carType {
  int  vehicleID;    /* Unique identifier for a car */
  char make[20];     /* Manufacturer                */
  char model[20];    /* Model name                  */
  int  year;         /* Year of manufacture         */
  int mileage;       /* in miles                    */
  double cost;       /* in dollars                  */

  Car *next;         /* Points to a car_node        */
};
```

Notice that this structure contains a pointer element that points to something of the same type as itself, or type `Car`. We will use this member element to point to the next node in the linked list. If the `next` field is equal to `NULL`, then the node is the last in the list.

```
int main()
{
  int op = 0;  /* Current operation to be performed.      */
  Car carBase; /* carBase an empty head node              */
               /*                                         */
  carBase.next = NULL;    /* Initialize the list to empty */

  printf("=========================\n");
  printf("=== Used car database ===\n");
  printf("=========================\n\n");

  while (op != 4) {
    printf("Enter an operation:\n");
    printf("1 - Car aquired. Add a new entry for it.\n");
    printf("2 - Car sold. Remove its entry.\n");
    printf("3 - Query. Look up a car's information.\n");
    printf("4 - Quit.\n");
    scanf("%d", &op);

    if (op == 1)
        AddEntry(&carBase);
    else if (op == 2)
        DeleteEntry(&carBase);
    else if (op == 3)
        Search(&carBase);
    else if (op == 4)
        printf("Goodbye.\n\n");
    else
        printf("Invalid option. Try again.\n\n");
  }
}
```

Figure 19.5 The function `main` for our used car database program

Now that we have defined the elementary data type and the organization of data in memory, we want to focus on the flow of the program, which we can do by writing the function `main`. The code is listed in Figure 19.5.

With this code, we create a menu-driven interface for the used car database. The main data structure is accessed using the variable `carBase`, which is of type `CarNode`. We will use it as a *dummy* head node, meaning that we will not be storing any information about any particular car within the fields of `carBase`; instead, we will use `carBase` simply as a placeholder for the rest of the linked list. Using this dummy head node makes the algorithms for inserting and deleting slightly simpler because we do not have to deal with the special case of an empty list. Initially, `carBase.next` is set equal to `NULL`, indicating that no data items are stored in the database. Notice that we pass the address of `carBase` whenever we call the functions to insert a new car in the list (`AddEntry`), to delete a car (`DeleteEntry`), and to search the list for a particular car (`Search`).

```
1  Car *ScanList(Car *headPointer, int searchID)
2  {
3    Car *previous;
4    Car *current;
5
6    /* Point to start of list */
7    previous = headPointer;
8    current = headPointer->next;
9
10   /* Traverse list -- scan until we find a node with a    */
11   /* vehicleID greater than or equal to searchID          */
12   while ((current != NULL) &&
13          (current->vehicleID < searchID)) {
14     previous = current;
15     current  = current->next;
16   }
17
18   /* The variable previous points to node prior to the    */
19   /* node being searched for. Either current->vehicleID   */
20   /* equals searchID or the node does not exist.          */
21   return previous;
22 }
```

Figure 19.6 A function to scan through the linked list for a particular vehicle ID

As we shall see, the functions `AddEntry`, `DeleteEntry`, and `Search` all rely upon a basic operation to be performed on the linked list: scanning the list to find a particular node. For example, when adding the entry for a new car, we need to know where in the list the entry should be added. Since the list is kept in sorted order of increasing vehicle ID numbers, any new car node added to the list must be placed *prior* to the first existing node with a larger vehicle ID. To accomplish this, we have created a support function called `ScanList` that traverses the list (which is passed as the first argument) searching for a particular vehicle ID (passed as the second argument). `ScanList` always returns a pointer to the node **just before** the node for which we are scanning. If the node we are scanning for is not in the list, then `ScanList` returns a pointer to the node **just prior** to the place in the list where the node would have resided. Why does `ScanList` return a pointer to the previous node? As we shall see, passing back the previous node makes inserting new nodes easier. The code for `ScanList` is listed in Figure 19.6.

Next we will examine the function to add a newly acquired car to the database. The function `AddEntry` gets information from the user about the newly acquired car and inserts a node containing this information into the proper spot in the linked list. The code is listed in Figure 19.7. The first part of the function allocates a `CarNode`-sized chunk of memory on the heap using `malloc`. If the allocation fails, an error message is displayed and the program exits using the `exit` library call, which terminates the program. The second part of the function reads in input from the standard keyboard and assigns it the proper fields within the new node. The third part performs the insertion by calling `ScanList` to find the place in the list to insert the new node. If the node already exists in the list then an error message is displayed and the new node is deallocated by a call to the `free` library call.

```
1  void AddEntry(Car *headPointer)
2  {
3    Car *newNode;       /* Points to the new car info       */
4    Car *nextNode;      /* Points to car to follow new one  */
5    Car *prevNode;      /* Points to car before this one    */
6
7    /* Dynamically allocate memory for this new entry.       */
8    newNode = (Car *) malloc(sizeof(Car));
9
10   if (newNode == NULL) {
11      printf("Error: could not allocate a new node\n");
12      exit(1);
13   }
14
15   printf("Enter the following info about the car.\n");
16   printf("Separate each field by white space:\n");
17   printf("vehicle_id make model year mileage cost\n");
18
19   scanf("%d %s %s %d %d %lf",
20         &newNode->vehicleID, newNode->make, newNode->model,
21         &newNode->year, &newNode->mileage, &newNode->cost);
22
23   prevNode = ScanList(headPointer, newNode->vehicleID);
24   nextNode = prevNode->next;
25
26   if ((nextNode == NULL) ||
27       (nextNode->vehicleID != newNode->vehicleID)) {
28     prevNode->next = newNode;
29     newNode->next = nextNode;
30     printf("Entry added.\n\n");
31   }
32   else {
33     printf("That car already exists in the database!\n");
34     printf("Entry not added.\n\n");
35     free(newNode);
36   }
37 }
```

Figure 19.7 A function to add an entry to the database

Let's take a closer look at how a node is inserted into the linked list. Figure 19.8 shows a pictorial representation of this process. Once the proper spot to insert is found using `ScanList`, first, the `prevNode`'s `next` pointer is updated to point to the new node and, second, the new node's `next` pointer is updated to point to `nextNode`. Also shown in the figure is the degenerate case of adding a node to an empty list. Here, `prevNode` points to the empty head node. The head node's `next` pointer is updated to point to the new node.

The routine to delete a node from the linked list is very similar to `AddEntry`. Functionally, we want to first query the user about which vehicle ID to delete and then use `ScanList` to locate a node with that ID. Once the node is found, the list is manipulated to remove the node. The code is listed in Figure 19.9. Notice that once

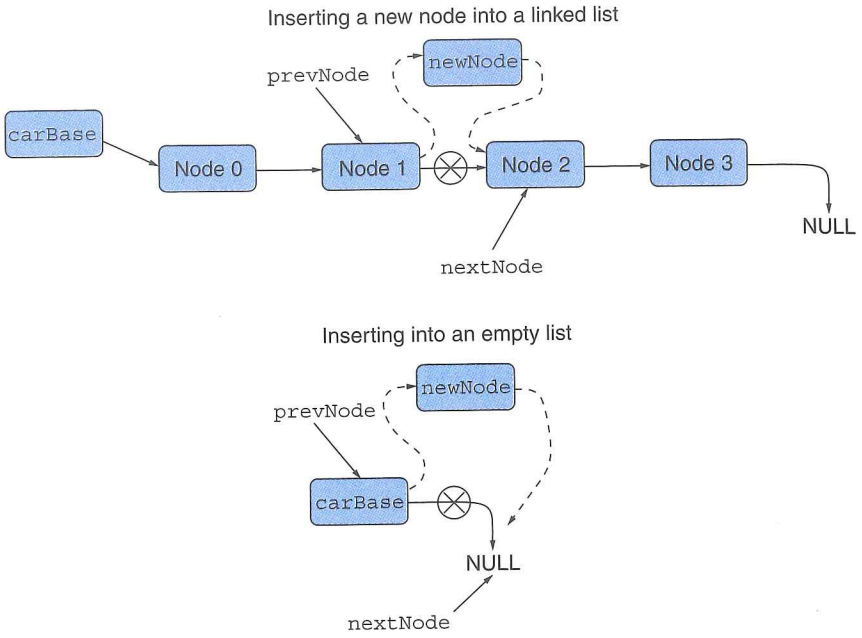


Figure 19.8 Inserting a node into a linked list. The dashed lines indicate newly formed links

```
1  void DeleteEntry(Car *headPointer)
2  {
3    int vehicleID;
4    Car *delNode;       /* Points to node to delete          */
5    Car *prevNode;      /* Points to node prior to delNode   */
6
7    printf("Enter the vehicle ID of the car to delete:\n");
8    scanf("%d", &vehicleID);
9
10   prevNode = ScanList(headPointer, vehicleID);
11   delNode  = prevNode->next;
12
13   /* Either the car does not exist or                       */
14   /* delNode points to the car to be deleted.               */
15   if (delNode != NULL && delNode->vehicleID == vehicleID) {
16     prevNode->next = delNode->next;
17     printf("Vehicle with ID %d deleted.\n\n", vehicleID);
18     free(delNode);
19   }
20   else
21     printf("The vehicle was not found in the database\n");
22 }
```

Figure 19.9 A function to delete an entry from the database

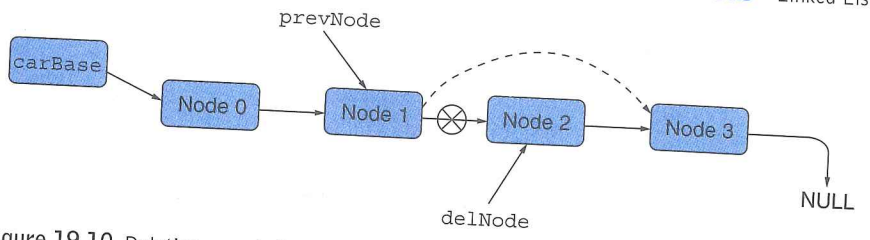


Figure 19.10 Deleting a node from a linked list. The dashed line indicates a newly formed link

```
1  void Search(Car *headPointer)
2  {
3    int vehicleID;
4    Car *searchNode;     /* Points to node to delete to follow */
5    Car *prevNode;       /* Points to car before one to delete */
6
7    printf("Enter the vehicle ID number of the car to search for:\n");
8    scanf("%d", &vehicleID);
9
10   prevNode = ScanList(headPointer, vehicleID);
11   searchNode  = prevNode->next;
12
13   /* Either the car does not exist in the list or                  */
14   /* searchNode points to the car we are looking for.              */
15   if (searchNode != NULL && searchNode->vehicleID == vehicleID) {
16     printf("vehicle ID : %d\n", searchNode->vehicleID);
17     printf("make       : %s\n", searchNode->make);
18     printf("model      : %s\n", searchNode->model);
19     printf("year       : %d\n", searchNode->year);
20     printf("mileage    : %d\n", searchNode->mileage);
21
22     /* The following printf has a field width specification on   */
23     /* %f specification.  The 10.2 indicates that the floating   */
24     /* point number should be printed in a 10 character field    */
25     /* with two units after the decimal displayed.               */
26     printf("cost       : $%10.2f\n\n", searchNode->cost);
27   }
28   else {
29     printf("The vehicle ID %d was not found in the database.\n\n",
30            vehicleID);
31   }
32 }
```

Figure 19.11 A function to query the database

a node is deleted, its memory is added back to the heap using the `free` function call. Figure 19.10 shows a pictorial representation of the deletion of a node.

At this point, we can draw an interesting parallel between the way elements are inserted and deleted from linked lists versus arrays. In a linked list, once we have identified the item to delete, the deletion is accomplished by manipulating

a few pointers. If we wanted to delete an element in an array, we would need to move all elements that follow it in the array upwards. If the array is large, this can result in a significant amount of data movement. The bottom line is that the operations of insertion and deletion can be cheaper to perform on a linked list than on an array.

Finally, we write the code for performing a search. The `Search` operation is very similar to the `AddEntry` and `DelEntry` functions, except that the list is not modified. The code is listed in Figure 19.11. The support function `ScanList` is used to locate the requested node.

19.6 Summary

We conclude this chapter by a summarizing the three key concepts we covered.

- **Structures in C.** The primary objective of this chapter was to introduce the concept of user-defined aggregate types in C, or structures. C structures allow us to create new data types by grouping together data of more primitive types. C structures provide a small step toward object orientation, that is, of structuring a program around the real-world objects that it manipulates rather than the primitive types supported by the underlying computing system.

- **Dynamic memory allocation.** The concept of dynamic memory allocation is an important prerequisite for advanced programming concepts. In particular, dynamic data structures that grow and shrink during program execution require some form of memory allocation. C provides some standard memory allocation functions such as `malloc`, `calloc`, `realloc`, and `free`.

- **Linked lists.** We combine the concepts of structures and dynamic memory allocation to introduce a fundamental new data structure called a linked list. It is similar to an array in that it contains data that is best organized in a list fashion. Why is the linked list such an important data structure? For one thing, it is a dynamic structure that can be expanded or shrunk during execution. This dynamic quality makes it appealing to use in certain situations where the static nature of arrays would be wasteful. The concept of connecting data elements together using pointers is fundamental, and you will encounter it often when dealing with advanced structures such as hash tables, trees, and graphs.

Exercises

19.1 Is there a bug in the following program? Explain.

```
struct node {
  int count;
  struct node *next;
};

int main()
{
  int data = 0;
  struct node *getdata;

  getdata->count = data + 1;
  printf("%d", getdata->count);
}
```

19.2 The following are a few lines of a C program:

```
struct node {
  int count;
  struct node *next;
};

main()
{
  int data = 0;
  struct node *getdata;

        :
        :

  getdata = getdata->next;

        :
        :
}
```

Write, in LC-3 assembly language, the instructions that are generated by the compiler for the line `getdata = getdata->next;`.

19.3 The code for `PotentialCollisions` in Figure 19.2 performs a pairwise check of all aircraft currently in the airspace. It checks each plane with every other plane for a potential collision scenario. This code, however, can be made more efficient with a very simple change. What is the change?

19.4 The following program is compiled on a machine in which each basic data type (pointer, character, integer, floating point) occupies one location of memory.

```
struct element {
  char  name[25];
  int   atomic_number;
  float atomic_mass;
};

is_it_noble(struct element t[], int i)
{
  if ((t[i].atomic_number==2)  ||
      (t[i].atomic_number==10) ||
      (t[i].atomic_number==18) ||
      (t[i].atomic_number==36) ||
      (t[i].atomic_number==54) ||
      (t[i].atomic_number==86))
    return 1;
  else
    return 0;
}

int main()
{
  int x, y;
  struct element periodic_table[110];

    :
    :
  x = is_it_noble(periodic_table, y);
    :
    :
}
```

a. How many locations will the activation record of the function `is_it_noble` contain?

b. Assuming that `periodic_table`, `x`, and `y` are the only local variables, how many locations in the activation record for `main` will be devoted to local variables?

19.5 The following C program is compiled into the LC-3 machine language and executed. The run-time stack begins at xEFFF. The user types the input `abac` followed by a return.

```
#include <stdio.h>
#define MAX 4

struct char_rec {
  char ch;
  struct char_rec *back;
};

int main()
{
  struct char_rec *ptr, pat[MAX+2];
  int i = 1, j = 1;

  printf("Pattern: ");
  pat[1].back = pat;
  ptr = pat;

  while ((pat[i].ch = getchar()) != '\n') {

    ptr[++i].back = ++ptr;

    if (i > MAX) break;
  }

  while (j <= i)
    printf("%d ", pat[j++].back - pat);

/* Note the pointer arithmetic here: subtraction
   of pointers to structures gives the number of
   structures between addresses, not the number
   of memory locations */
}
```

a. Show the contents of the activation record for `main` when the program terminates.

b. What is the output of this program for the input `abac`?